UNIVERSITÉ CÔTE D'AZUR
Oser créer

Natural Language Processing: Project Report

Log Analyzer

Supervisor : Cabrio Elena

Année 2024/2025

Master 1 Informatique
Benoît Barbier
Massinissa Abboud

RAPPORT

1. Project Context and Task

System logs are ubiquitous in software systems, recording runtime events and traces of program execution. They enable tasks such as troubleshooting failures and detecting anomalies by providing a historical record of system behavior (Nedelkoski et al., 2020). However, raw logs are semi-structured text: each line is a message that intermixes constant parts (the static message template) with variable parts (parameters like identifiers, timestamps, etc.). Parsing these semi-structured records into structured log templates (with placeholders for the variable fields) is an essential first step to make logs analyzable (Nedelkoski et al., 2020). A log template (also called an event template or log key) represents the static portion of a log message, abstracting away specifics like unique IDs or addresses. For example, many lines may share a template "Received block <*> of size <*> from <*>" while the placeholders <*> differ per line (e.g., different block IDs, sizes, and IP addresses).

Extracting log templates automatically (i.e. log parsing) is challenging. Traditional log parsers rely on domain-specific heuristics or manually crafted rules to distinguish constant vs. variable parts (Nedelkoski et al., 2020). These methods often need tuning for each log format and struggle to generalize across diverse systems. In this project, we formulate log template extraction as a self-supervised learning problem using a masked language model (MLM). Masked language modeling is well-suited because it learns to predict missing tokens from context, which naturally aligns with identifying fixed patterns in logs. Essentially, if we mask out a token in a log message, an MLM can leverage surrounding words to decide if that token is predictable (indicating it was part of the common template) or if it behaves like a random value (indicating a variable parameter). NuLog, a recent approach by Nedelkoski *et al.* (2020), demonstrated this idea: *"Unlike traditional methods that rely on heuristics or manual rule extraction, NuLog uses masked language modeling (MLM) to predict the presence of words in log messages based on their context, distinguishing between constant and variable parts without requiring domain knowledge."* (Hashemi & Mäntylä, 2024). This data-driven approach does not require labeled templates for training; instead, the model learns the log structure from raw logs alone (self-supervision). By learning contextual patterns, a masked LM can generalize across log types and adapt to varied message formats, which is the core hypothesis of our project.

2. Dataset Description

We evaluate our approach on the NuLog benchmark dataset (NuLog Team, n.d.), which contains logs collected from 16 diverse software systems. These include distributed systems (e.g., Hadoop HDFS and Spark), supercomputer logs (e.g., Blue Gene/L (BGL), HPC cluster logs, Thunderbird), operating system logs (Windows, Linux, Mac OS), mobile applications (Android, HealthApp), server applications (Apache HTTP server, OpenSSH).. Each system's dataset consists of 2,000 raw log messages and an accompanying template file with the ground truth log templates (as produced by a state-of-the-art rule-based parser). In total, the corpus comprises 32,000 log lines and their corresponding templates. Every log line in a given file is labeled with a template identifier that links to a template (with wildcards for variables) in the template file. We utilize the raw log messages for model training and the template files for evaluation, i.e., to check how well the model's extracted templates match the known ground truth.

To organize experiments, we first concatenated the log files from all 16 systems into a single corpus (32 000 lines total) and then performed a straightforward 90 %/10 % split at the line level:

- Training set**:** 28 800 lines (90 %), used both for fitting the masked LM and for hyper-parameter tuning via internal cross-validation.
- Test set**:** 3 200 lines (10 %), held out entirely until the final evaluation step.

Because the split is random across the combined corpus, each set contains a representative mix of formats, from highly repetitive logs such as Apache server errors (only 6 unique templates in 2 000 lines) to extremely diverse sources like Mac OS logs (342 templates in 2 000 lines) (Zhu, 2019). This wide range of template diversity, now uniformly present in both training and test data, reinforces the need for a robust, general parsing approach.

To illustrate the format diversity, consider a sample from the HDFS log vs. one from Windows Event Log. An HDFS message (raw log) might be:

INFO Receiving block 12345 src: /10.1.1.5 dest: /10.2.2.8

with the ground truth template:

Receiving block <> src: <> dest: <*>*

Here the static structure ("Receiving block _ _ src: _ dest: _") is constant across many lines, while the numeric block ID and two IP addresses vary per line. In contrast, a Windows Event Log entry could look like:

Information 2021-11-01 12:00:00 PM ServiceControlManager 7036: The SomeService service entered the running state.

The template for this might abstract away the timestamp and dynamic data as:

The <> service entered the <*> state.*

This shows how logs can have very different formats (Linux/Unix-style vs. Windows-style in this case) and how templates generalize them by removing specifics. The model must learn to handle such differences in syntax, punctuation, and vocabulary across the dataset.

3. Methodology

Our approach to log template extraction consists of four main components:

1. Log Tokenization: Converting raw log messages into sequences of tokens

2. Dictionary Building: Creating a vocabulary of tokens from template files
3. BERT Model Training: Fine-tuning a BERT model using masked language modeling
4. Template Extraction: Using the trained model to identify template and variable parts

3.1 Hashing-based Tokenization

To train a masked language model on log data, we first need to convert each raw log message into a sequence of token IDs. A straightforward approach of treating every unique word or symbol as a distinct vocabulary token is infeasible here – the logs contain myriad unique identifiers (memory addresses, UUIDs, timestamps, etc.), which would explode the vocabulary size. Instead, we designed a hashing-based tokenization method to constrain the vocabulary. The key idea is to map tokens that are different in raw text into a limited set of numeric IDs, intentionally allowing collisions. This trades off some token uniqueness for a much smaller fixed vocabulary that the model can learn. By hashing, we ensure the model focuses on the general patterns rather than memorizing every distinct value.

Token categorization**:** Each token (word or number separated by whitespace in a log line) is classified into one of several predefined types before hashing. We defined the following token types, each of which will be mapped into a designated ID range:

IP addresses: Tokens matching an IP address pattern (e.g. 192.168.10.5) are detected via a regex (four numeric octets). These are hashed into a specific range reserved for IPs.

Email addresses: Tokens containing an "@" (e.g. user@example.com) are treated as emails and hashed into a range for email tokens.

Date/Time strings: Tokens that appear to be timestamps or date strings (e.g. 2021-09-15 or 12:45:30) are mapped to a date/time range. We include both purely numeric dates and common datetime formats in this category.

Large numeric values: Numeric tokens that likely represent IDs, memory addresses, or other large numbers (for example, 64821934 or 0x7FFA9C30) fall into this category. We chose a threshold (e.g., length > 4 digits) to differentiate large numbers from small constants. These are hashed to an numeric-ID range.

Words: Alphabetic tokens (consisting only of letters, like ERROR, failed, Connection) are hashed into the general "word" vocabulary range. This covers most constant message text.

Unknown/Other: Any token that doesn't fit the above (for instance, tokens with mixed alphanumeric characters or special symbols, such as file paths C:\Windows\... or a string like errCode123) defaults to an "unknown" category. This ensures every token gets categorized. Unknown tokens are hashed into their own range.

Each category is assigned a distinct ID range in the vocabulary. For example, we might reserve ID 0–999 for IPs, 1000–1099 for emails, and so on (exact ranges tuned based on expected variety per type). The hash_token() function then works as follows: it takes a raw token string, identifies its type, and computes a hash value (using a stable hash function) which is modulo the size of that type's range. The final token ID is the hash value offset by that category's start index. This design ensures that no token from one category can collide with a token from another category (e.g., an IP address will never share an ID with a word or a number). However, within each category, many different raw tokens may map to the same hashed ID – these are the collisions we allow in exchange for vocabulary compression.

Token sequence conversion: Using this hashing scheme, we implemented a function log_to_token_seq() to transform a raw log line into a sequence of token IDs. The procedure is:

1. Split the log message by whitespace into tokens. For example, the HDFS line INFO Received block 12345 from 10.1.1.5 splits into ["INFO", "Received", "block", "12345", "from", "10.1.1.5"]. (Additional preprocessing can be applied here, such as removing punctuation or isolating certain symbols, but in our implementation we primarily use whitespace splitting for simplicity.)

2. Hash each token using the rules above. For each token, hash_token() assigns it a category and produces a numeric ID in that category's range. For instance, "INFO" is all letters so it would be hashed in the word range (suppose it becomes ID 4501), "12345" is a number (numeric range, say ID 3005), and "10.1.1.5" is an IP (IP range, say ID 42).

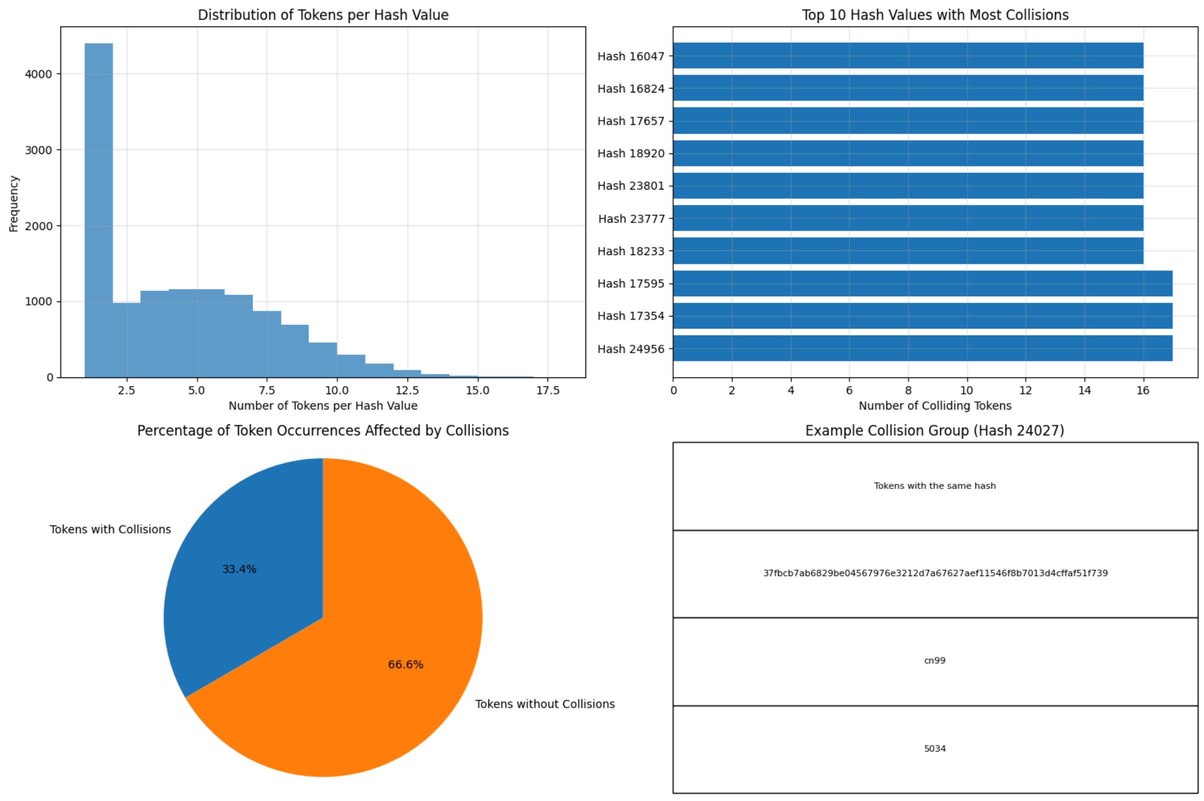
3. Return the ID sequence. The log line is now represented as an integer sequence, e.g. [4501, 4607, 4778, 3005, 4621, 42] (this is just an illustrative sequence of IDs corresponding to the tokens above). These ID sequences form the input to our masked language model. During training, we randomly mask some of these IDs and train the model to predict them, following the standard MLM objective.

This hashing-based tokenization dramatically reduces the vocabulary size. Instead of potentially hundreds of thousands of unique tokens across the 32k log lines, we might fix the vocabulary to a few thousand hashed tokens. The masked LM therefore learns on a manageable vocabulary that still roughly captures token distinctions like "this is some IP address" or "this is some word," without getting bogged down by exact values.

Collision analysis**:** An inevitable side effect of hashing is that different raw tokens can collide to the same ID. We analyzed the extent of collisions in our tokenization by examining how many distinct raw tokens map to each hash ID. **Figure 1** below shows the distribution of these collisions across our dataset.

Most hash values (~4,300) have exactly one token mapping to them, shown by the tall bar at x=1. However, a significant number of hash values experience collisions, with some having 2, 3, or more different tokens all mapping to the same hash ID. The histogram reveals a long tail extending past x=15, indicating some extreme cases where many distinct tokens collide to the same hash value. In total, approximately 33.4% of token occurrences in our logs are affected by collisions, meaning that about one-third of the time, a token appears with an ID that is shared with at least one other distinct token. This is visualized in the pie chart, which shows the proportion of token occurrences affected by collisions versus those that map uniquely.

The top 10 worst collision cases (shown in the top-right chart) each have 16 or more different tokens mapping to a single hash value. For example, hash 24027 (bottom-right table) maps together three completely different tokens: a long hexadecimal string, "cn99", and "5034".



Tokens with the same hash
37fbcb7ab6829be04567976e3212d7a67627aef11546f8b7013d4cffaf51f739
cn99
5034

Figure 1: Distribution of tokens per hash value in our dataset. The histogram shows how many hash values (y-axis: Frequency) have a specific number of tokens mapping to them (x-axis: Number of Tokens per Hash Value). A value of 1 on the x-axis means no collision (one unique token maps to that hash), while values of 2 or higher indicate collisions.

These collisions are not uniformly distributed across token types. Certain categories like alphabetic words and unknown tokens (which catch varied strings like mixed IDs, paths, etc.) show higher collision rates. This suggests these token types have much greater diversity in the data than our hashing space allocated to them. In contrast, categories like email tokens show fewer collisions, likely because they appear less frequently or with less variety in the logs.

Despite these collisions, the masked language model can still effectively learn templates because the overall patterns of which token positions are static versus variable tend to emerge from the data, even if some token identities are conflated. The model mitigates collision impact by using surrounding context tokens and by structuring hash ranges so that common tokens are less likely to collide.

3.2 Vocabulary

A vocabulary of log template tokens was constructed from the EventTemplate field of log data collected from 16 different log sources (NuLog dataset). Each log template (a message pattern with variable fields) was tokenized using a custom split_log() function. This function applies a series of regular expression patterns to extract meaningful tokens, for example, matching IP addresses, timestamps, words, and numeric sequences as whole units. By using these regex rules, structured elements like 192.168.0.1 or date-time strings are captured as single tokens rather than being split into arbitrary pieces. All tokens extracted from the static (constant) parts of every template were aggregated, and the placeholder token <*> (used in templates to denote variable content) was excluded from the vocabulary. The result is a set of unique tokens that appear in the fixed text of log templates across the 16 sources, representing the core vocabulary used for log parsing.

The vocabulary size varies widely across the different log sources. A bar chart of vocabulary size per source (Figure 2) shows that certain systems have significantly larger vocabularies than others. In particular, the Mac OS system logs and the Thunderbird application logs exhibit markedly higher numbers of unique tokens compared to other sources. This implies that these systems produce a far more diverse set of log templates or messages, resulting in a larger variety of distinct terms. In contrast, some other log sources (e.g., those from simpler or more standardized applications) have much smaller vocabularies, indicating more repetitive or uniform log messages. The considerable disparity in vocabulary size highlights how the complexity and variability of logged events can differ between software systems.

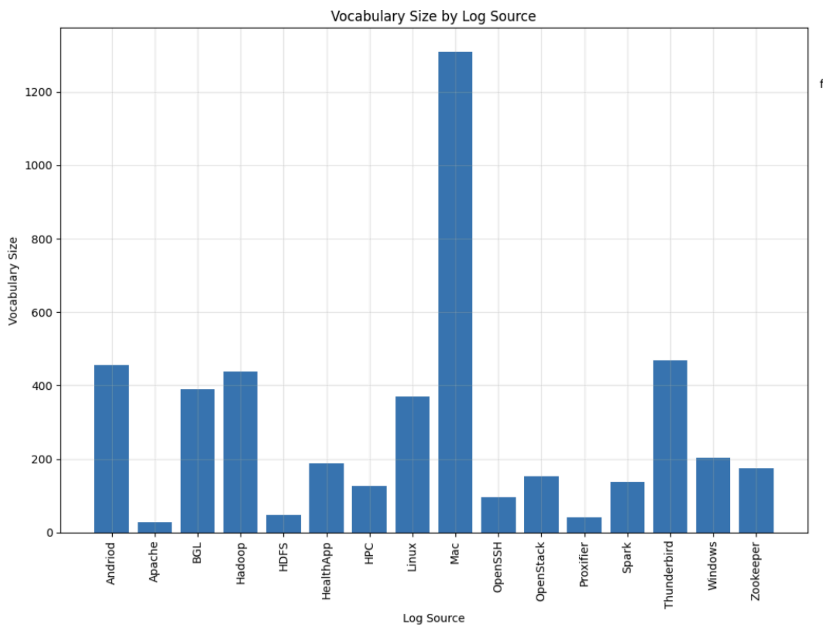
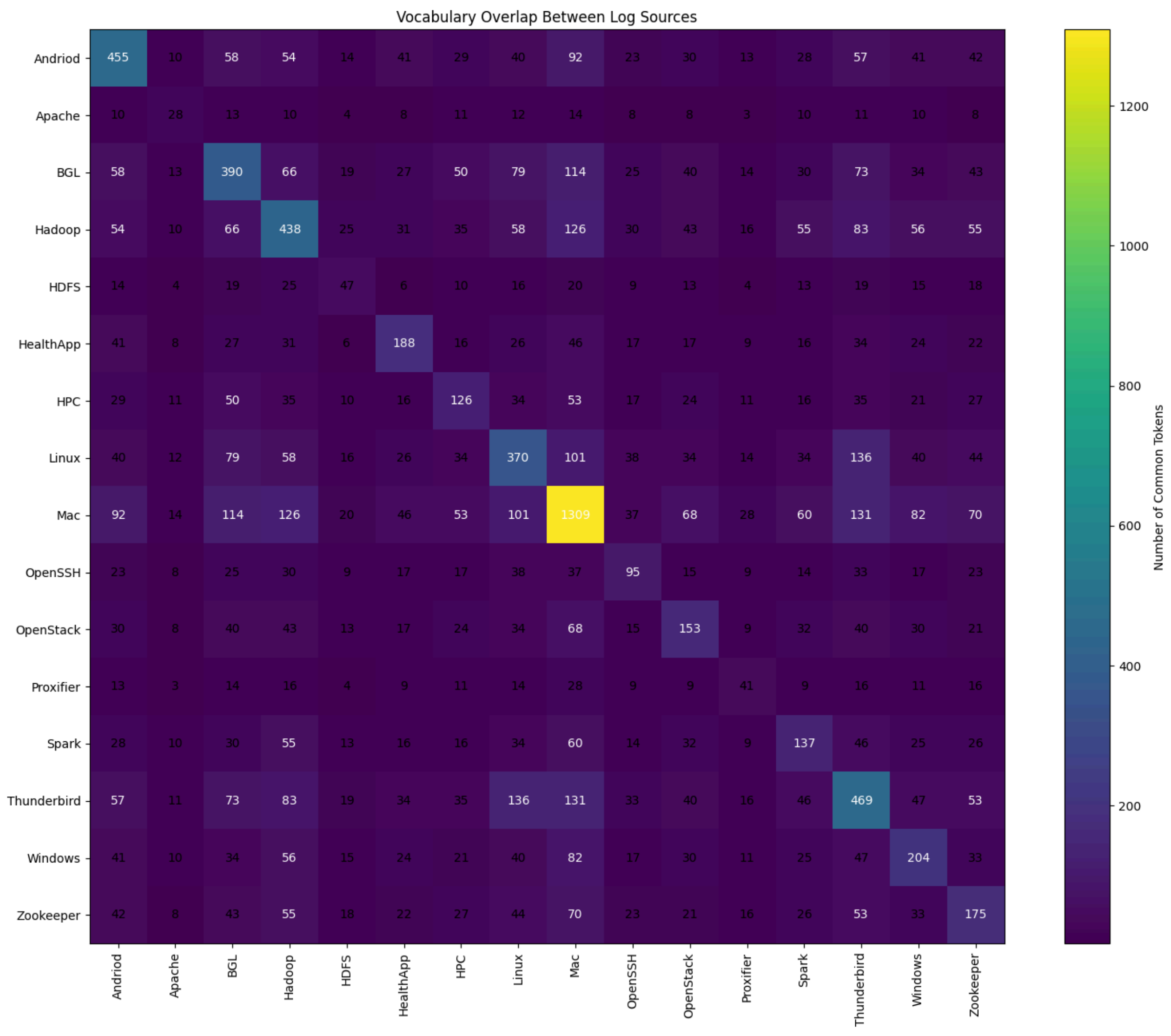


Figure 2: Vocabulary size per log source

To assess commonality between log sources, a vocabulary overlap heatmap (Figure 3) was generated, illustrating the fraction of tokens shared by each pair of sources. The heatmap reveals that there is only limited overlap in vocabulary between most log sources. Notably, a moderate degree of token overlap appears among a few related systems, for instance, the Mac, Linux, and Thunderbird logs share some common tokens, reflecting overlapping terminology (perhaps generic error messages or OS-related terms). However, for the majority of source pairs, the overlap is minimal or near zero, which is evident from the predominantly low off-diagonal values in the heatmap. In other words, each system's log vocabulary is largely distinct. This indicates that different software platforms and applications tend to use their own domain-specific terms in log messages, with only a small core of general log terms (e.g. common status or error keywords) appearing across multiple sources.



	Andriod	Apache	BGL	Hadoop	HDFS	HealthApp	HPC	Linux	Mac	OpenSSH	OpenStack	Proxifier	Spark	Thunderbird	Windows	Zookeeper
Andriod	455	10	58	54	14	41	29	40	92	23	30	13	28	57	41	42
Apache	10	28	13	10	4	8	11	12	14	8	8	3	10	11	10	8
BGL	58	13	390	66	19	27	50	79	114	25	40	14	30	73	34	43
Hadoop	54	10	66	438	25	31	35	58	126	30	43	16	55	83	56	55
HDFS	14	4	19	25	47	6	10	16	20	9	13	4	13	19	15	18
HealthApp	41	8	27	31	6	188	16	26	46	17	17	9	16	34	24	22
HPC	29	11	50	35	10	16	126	34	53	17	24	11	16	35	21	27
Linux	40	12	79	58	16	26	34	370	101	38	34	14	34	136	40	44
Mac	92	14	114	126	20	46	53	101	1309	37	68	28	60	131	82	70
OpenSSH	23	8	25	30	9	17	17	38	37	95	15	9	14	33	17	23
OpenStack	30	8	40	43	13	17	24	34	68	15	153	9	32	40	30	21
Proxifier	13	3	14	16	4	9	11	14	28	9	9	41	9	16	11	16
Spark	28	10	30	55	13	16	16	34	60	14	32	9	137	46	25	26
Thunderbird	57	11	73	83	19	34	35	136	131	33	40	16	46	469	47	53
Windows	41	10	34	56	15	24	21	40	82	17	30	11	25	47	204	33
Zookeeper	42	8	43	55	18	22	27	44	70	23	21	16	26	53	33	175

Figure 3: Vocabulary overlap between log sources

3.4 Model

In this section, we explore the use of BERT for Masked Language Modeling (MLM) to classify tokens in logs as either *static* or *variable*. The central idea is to train a MLM model to predict token behavior in a way that reflects its occurrence and importance within the log data. Tokens that are static are more predictable, while variable tokens are less consistent in their patterns, making them harder for the model to predict.

Initially, we attempted to use MLM confidence to classify tokens by leveraging two approaches: (1) entropy on the model's logits and (2) checking token prediction accuracy in the top-k predictions. However, this approach was not sufficiently robust for confident token classification. Nonetheless, the model showed a clear trend where static tokens were predicted with higher confidence, and they appeared more often in the top-k predictions.

To improve classification performance, we transitioned to a supervised approach, fine-tuning the model with an added token classification head, specifically designed to classify tokens as static or variable.

3.4.1 Base Model – BERT for Masked Language Modeling (MLM)

Unlike standard practice where BERT is initialized from a publicly available pre-trained checkpoint, we trained a BERT model from scratch using our own log data. This decision was motivated by the highly domain-specific nature of log files, which differ significantly from typical natural language text. As such, we found it more effective to train a model tailored specifically to our structured, repetitive, and often templated input format.

To train the model, we followed the Masked Language Modeling (MLM) objective. For this task, each log line was first tokenized using our custom tokenizer, which ensures that tokens reflect the structural and semantic patterns commonly found in logs. Then, for each tokenized log line, we generated training samples by randomly masking one token at a time, creating training batches where each data point corresponds to a single masked token in context.

For example, from a single tokenized log line [token_1, token_2, ..., token_n], we create *n* training examples by masking one token at a time (e.g., [token_1, [MASK], token_3, ..., token_n]) and asking the model to predict the masked token. This setup ensures that the model learns contextual patterns around every token position in a log line.

The BERT model was trained on these masked token prediction tasks over our full log dataset. This allowed it to learn the underlying structure of logs, including common token sequences, fixed patterns, and the distribution of variable fields. The training objective was the standard cross-entropy loss over the vocabulary for the masked token position.

This custom-trained BERT model served as the foundation for our further experiments, including evaluating token confidence and building a supervised classifier.

3.4.2 Token Classification Approach

To classify tokens as static or variable, we used two initial approaches based on the MLM's logits:

1. Entropy on Logits**:** We used the entropy of the model's logits as a confidence measure. The idea was that static tokens would yield lower entropy (i.e., the model would be more confident in predicting them), while variable tokens would result in higher entropy due to the model's uncertainty.

2. Top-k Token Prediction**:** In this approach, we masked each token in the sequence and used the MLM model to predict the masked token. If the token appeared in the top-k predictions, we classified it as *static*, as the model was highly confident in predicting it correctly. Otherwise, it was classified as *variable*.

Despite observing that the entropy was generally lower for static tokens and that static tokens were predicted more accurately (appearing in the top-k predictions 7 times more often than variable tokens), the results were not sufficient to confidently classify tokens.

3.5 Training Setup

To pre-train the Masked Language Model (MLM), we used a lightweight BERT architecture tailored for efficiency on domain-specific logs. The model was configured with the following parameters:

- Vocabulary size: 30,000
- Maximum sequence length: 300 tokens
- Number of transformer layers: 2
- Hidden size: 64
- Intermediate (feed-forward) size: 256
- Number of attention heads: 1
- Type vocab size: 1 (single-segment input)

The model was trained using the Hugging Face Trainer API with the following training settings:

- Batch size: 64 (for both training and evaluation)
- Number of epochs: 4
- Learning rate: 5e-5
- Weight decay: 0.01
- Evaluation strategy: every *n* steps (with best model loaded at the end)
- Logging: every 10 steps
- Model checkpointing: every 500 steps

We were constrained to downscale the size of the model in order to reduce the training time to a reasonable amount.

3.6 Fine-Tuning for Token Classification

In light of the previous method's limitations, we decided to move toward a more supervised approach by adding a classification head on top of the MLM model. This classification head was designed to predict whether each token was *static* or *variable*. The model was fine-tuned on a dataset of logs with token-level labels (static/variable) for more accurate and confident classification.

We fine-tuned the model by training it on a custom dataset that was tokenized using our custom tokenizer. The dataset contains log entries with labels indicating whether each token is static or variable. We used cross-entropy loss for training and optimized the model using AdamW optimizer.

Model Architecture: The architecture includes the following layers:

- BERT (Pre-trained MLM model)**:** This layer is used to capture the context of tokens in a sequence.
- Classification Head**:** A linear layer with a softmax activation function that outputs a probability for each token belonging to one of the two classes: *static* or *variable*.

Training Setup:

- Optimizer**:** AdamW optimizer was used for fine-tuning the model.
- Learning Rate**:** The learning rate used was: **2e-5**
- Batch Size**:** A batch size of 16 was used.
- Epochs**:** The model was trained for 20 epochs.

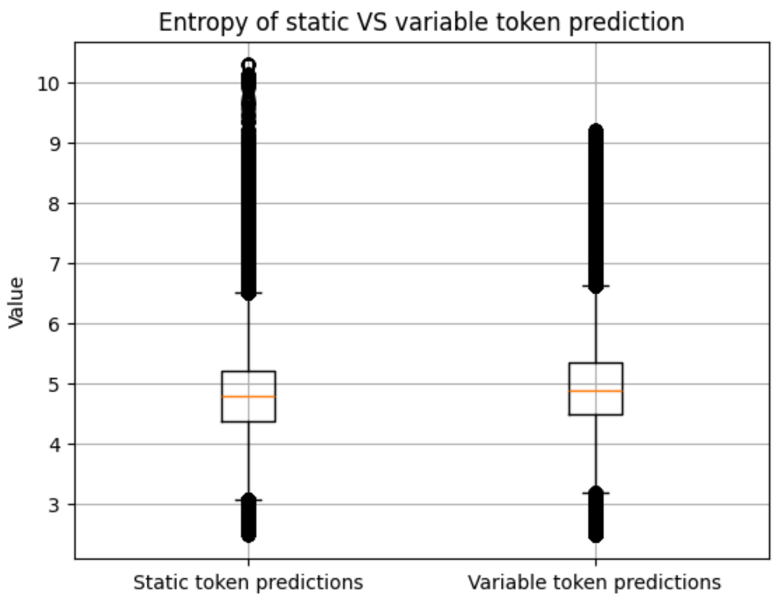
4. Evaluation

4.1 Evaluating the BERT MLM

To assess the quality of the trained Masked Language Model (MLM), we performed a detailed analysis of the model's predictive behavior on both *static* and *variable* tokens.

4.1.1 Entropy Analysis

We computed the entropy of the predicted token distribution at each masked position. A lower entropy indicates that the model is more confident in its prediction, while a higher entropy suggests uncertainty.
The box plot below shows the distribution of entropies for static and variable tokens. On average, static tokens exhibit significantly lower entropy, indicating that the model is more confident and consistent when predicting them. Conversely, the higher entropy for variable tokens reflects the model's greater difficulty in modeling these more dynamic or less frequent patterns.

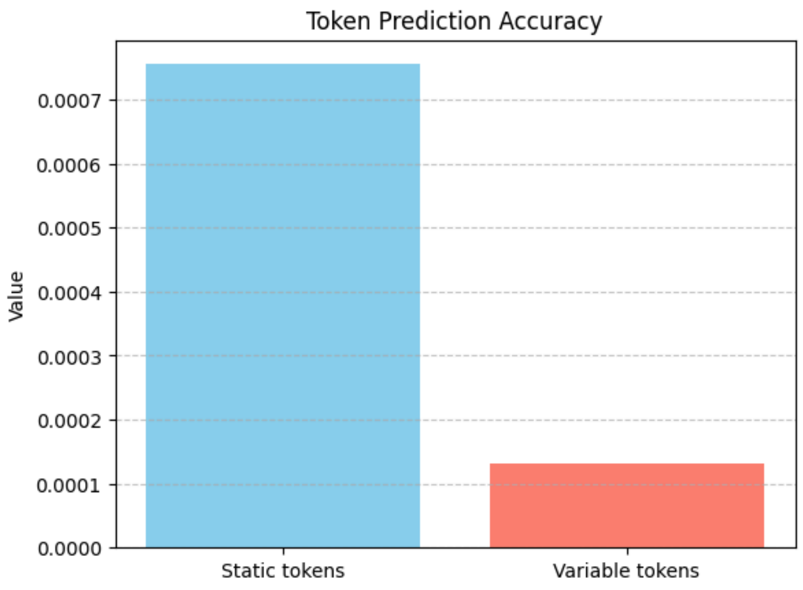


4.1.2 Top-k Prediction Accuracy

We further analyzed the model's ability to recover the correct token within its top-k predictions (with $k = 5$). The results are shown in the bar plot below.

The model is 7 times more likely to correctly predict static tokens within its top-5 guesses compared to variable tokens. This reinforces the conclusion from the entropy analysis: static tokens are more learnable and consistently predicted, while variable tokens present a greater challenge.

These results highlight the model's capacity to internalize recurring structures in logs (static tokens), while pointing to opportunities for improvement in handling more dynamic elements (variable tokens).



4.1.3 Conclusion

While these results confirm that the MLM is more confident and accurate when predicting static tokens, this distinction alone is not sufficient to reliably classify tokens into static or variable categories. The observed differences are encouraging but not decisive.

Hence, there is a clear need to fine-tune the model specifically for classification, enabling it to explicitly learn the static/variable distinction through supervised training.

4.2 Evaluating the Token Classificator Model

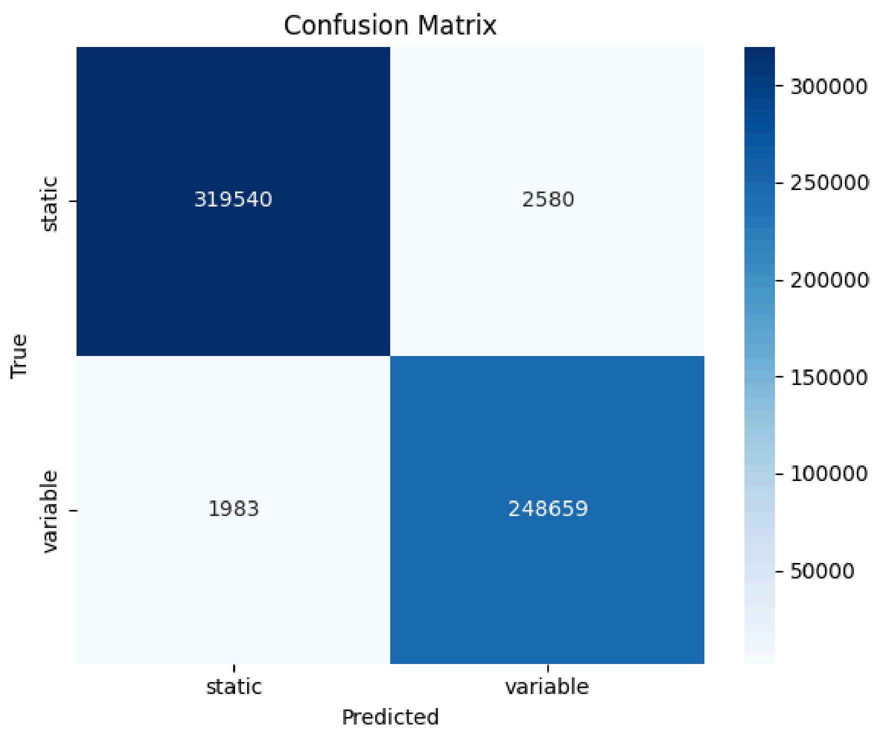
Following the evaluation of the MLM's predictive confidence, we fine-tuned the BERT model on a supervised classification task to explicitly distinguish between static and variable tokens.

4.2.1 Classification Metrics

The classifier achieves outstanding performance, with precision, recall, and F1-score all reaching 0.99 for both classes. These metrics demonstrate that the model is highly effective at distinguishing between the two categories, with near-perfect balance and no indication of bias toward one class over the other.

4.2.2 Confusion Matrix

The confusion matrix further confirms the classifier's strength:



Only a small fraction of tokens are misclassified. Specifically, about 0.8% of static tokens are incorrectly predicted as variables, and less than 0.8% of variable tokens are predicted as static. This tight diagonal structure highlights the model's reliability and fine-tuning effectiveness.

5. Error Analysis

To better understand the limits of our classifier, we analyzed the small subset of misclassified tokens. Despite the overall high accuracy, a closer look reveals insightful patterns of failure.

5.1 Confusion Near Class Boundaries

Some misclassifications occur on tokens that lie near the boundary between static and variable categories. These often include tokens that appear in both contexts, for example, common codes or identifiers that are templated in some logs but variable in others. This overlap creates semantic ambiguity that challenges even a well-trained model.

5.2 Rare Tokens and Token Overlap

Another source of error involves rare variable tokens, especially those whose subword composition resembles that of frequent static tokens. BERT's reliance on subword embeddings can lead to misclassifications when unfamiliar tokens share segments with more common counterparts. In such cases, the model may generalize incorrectly based on token overlap rather than true contextual usage.

5.3 Token Format Bias and Overfitting

The tokenizer design embeds useful clues in the token structure itself, for instance, IP addresses, email-like strings, or dates tend to appear as distinct tokens and are usually variable. The model learns these patterns effectively, associating certain token formats with specific classes. However, this becomes a double-edged sword: in cases where such formatted data appears within the static template (e.g., hardcoded email domains or fixed timestamps), the model is prone to overfit to token form rather than context. This highlights the model's tendency to rely heavily on token surface features, which may not always be reliable indicators of class membership.

6. Conclusion and Possible Improvements

This work demonstrated that it is possible to effectively distinguish between static and variable tokens using a lightweight BERT-based model fine-tuned for token classification. While our classifier achieves excellent performance on the test set, our analysis reveals both limitations and areas for further refinement.

A potential direction to reduce the reliance on the classifier altogether would be to further improve the Masked Language Model (MLM) itself. Training the MLM for more epochs, on a larger and more diverse dataset, and with a larger model architecture may enhance its ability to distinguish static from variable tokens directly based on prediction confidence or entropy. This would make the system more elegant and potentially more generalizable.

Another open question is how well the model generalizes to unseen log formats. Both the MLM and the classifier were trained on a specific dataset, and their performance on logs from different sources remains untested. This calls for more robust evaluation across varied domains.

Additionally, several improvements could address the sources of error discussed earlier. For example, mitigating overfitting to token format might require rethinking how tokens are represented or incorporating regularization strategies. Incorporating contextual sequence labels, such as using BIOS tagging schemes from Named Entity Recognition (NER), could also help the model better understand token roles within the broader structure of the log.

In summary, while the current approach shows strong promise, there are clear paths toward improving generalization, robustness, and interpretability, all important for deploying such models in real-world log analysis systems.

BIBLIOGRAPHIE

Nedelkoski, S., Bogatinovski, J., Acker, A., Cardoso, J., & Kao, O. (2020). *Self-Supervised Log Parsing*(arXiv:2003.07905). arXiv. https://doi.org/10.48550/arXiv.2003.07905

Hashemi, S., & Mäntylä, M. (2024). *Token Interdependency Parsing (Tipping) – Fast and Accurate Log Parsing*(arXiv:2408.00645). arXiv. https://doi.org/10.48550/arXiv.2408.00645

NuLog Team. (n.d.). *NuLog* [Computer software]. GitHub. https://github.com/nulog/nulog

Zhu, J., He, S., Liu, J., He, P., Xie, Q., Zheng, Z., & Lyu, M. R. (2019). *Tools and benchmarks for automated log parsing*. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)* (pp. 121–130). IEEE. https://doi.org/10.1109/ICSE-SEIP.2019.00023